Simple Seed Architectures for Reciprocal and Square Root Reciprocal

Milos Ercegovac, Jean-Michel Muller, Arnaud Tisserand

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Simple Seed Architectures for Reciprocal and Square Root Reciprocal

Milos Ercegovac, Jean-Michel Muller and Arnaud Tisserand

N° 5720

October 2005

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5720--FR+ENG

Simple Seed Architectures for Reciprocal and Square Root Reciprocal

Milos Ercegovac, Jean-Michel Muller and Arnaud Tisserand

Thème SYM — Systèmes symboliques
Projet Arénaire

Rapport de recherche n° 5720 — October 2005 — 25 pages

Abstract: This report presents a simple hardware architecture for computing the seed values for reciprocal and square root reciprocal. These seeds are used in the initialization of floating-point division and square root software iterations. The proposed solution is based on polynomial approximation with specific coefficients and a table lookup. The obtained architectures lead to small and fast circuits.

Key-words: computer arithmetic, reciprocal, square root reciprocal, polynomial approximation, table based method, hardware arithmetic operator, Newton-Raphson iteration

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Approximations initiales pour l'inverse et la racine carrée inverse

Résumé : Ce rapport présente une architecture simple de calcul d'approximations initiales pour l'inverse et la racine carrée inverse. Ces approximations initiales sont utilisées comme point de départ de la division et la racine carrée flottante en logiciel. La solution proposée est basée sur une approximation polynomiale avec des coefficients particuliers et une lecture dans une table. Les architectures obtenues permettent de réaliser des circuits petits et rapides.

Mots-clés : arithmétique des ordinateurs, inverse, racine carrée inverse, approximation polynomiale, méthode à base de tables, opérateurs matériels, itération de Newton-Raphson

Simple Seed Architectures for Reciprocal and Square Root Reciprocal

M. Ercegovac[1], J.-M. Muller[2] and A. Tisserand[3,2]

[1]University of California, Los Angeles
Computer Science Department
4732 Boelter Hall, Los Angeles, CA 90095, USA
milos@cs.ucla.edu

[2]CNRS, LIP (CNRS–ENS Lyon–INRIA–UCB Lyon)
46 allée d'Italie. F-69364 Lyon, FRANCE
jean-michel.muller@ens-lyon.fr

[3]CNRS, LIRMM
161 rue Ada. F-34392 Montpellier, FRANCE
arnaud.tisserand@lirmm.fr

Introduction

In general-purpose processors, floating-point division and square root are often performed using iterative methods such as the Newton-Raphson algorithm [7, 4, 6]. In order to reduce the number of iterations, dedicated hardware tables are frequently used to store medium accuracy initial values, or seeds, for the iterations. In this work, we focus on the computation of such seeds in hardware.

The proposed solution is based on a polynomial approximation with specific coefficients and a table lookup. The corresponding architecture is very simple and leads to small and fast circuits. There are several parameters in seed hardware operator: the approximated function ($1/x$ or $1/\sqrt{x}$), the argument width, internal accuracy requirements and optimization parameters. We have developed a VHDL code generator, called `seedgen`, to support all the possible parameters of our method. This program generates an optimized and synthesizable VHDL description of the seed operator based on our method. Compared to direct lookup table implementation on FPGAs (with content optimization using the synthesis tools), the proposed method leads to 2.5 reduction factor in size and 40% speed improvement.

This document is organized as follows. The background on division and square root iterations using the Newton-Raphson algorithm is presented in Section 1. Our solution for

the computation of the reciprocal seeds is presented in Section 2. The case of the square root reciprocal is presented in Section 3. In Section 4 we discuss the main characteristics of a tool, called `seedgen`, that generates VHDL descriptions of hardware seed generators based on our approach. The main results related to area and delay of implementations on the Xilinx Spartan3 and VirtexII FPGAs are presented in Section 5.

1 Background

1.1 Newton-Raphson Iteration

The Newton-Raphson algorithm evaluates a function by iteratively improving an initial approximation. This algorithm is based on a general method to obtain a single zero α of function f (i.e., $f(\alpha) = 0$ and $f'(\alpha) \neq 0$). If x_0 is close enough to α, the following iteration converges towards α:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{1}$$

where $f'(x)$ denotes the derivate of f with respect to x. The convergence of the method is quadratic, which means that the number of bits of accuracy roughly doubles after each iteration.

In order to speed up the overall computation, the first iterations that only provide a very small number of bits of accuracy should be avoided. For instance, if one uses an initial value x_0 with only one bit of accuracy, the computation requires at least 5 iterations ($1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32$) for 32-bit values. Using an initial seed with 8 bits of accuracy, it only requires 2 iterations ($8 \rightarrow 16 \rightarrow 32$).

In general, by using an initial approximation with a relative error not greater than 2^{-n}, the number of iterations to obtain the result with a relative error not greater than 2^{-m} is

$$K = \left\lceil log_2 \left(\frac{m}{n} \right) \right\rceil \tag{2}$$

Therefore, the choice of the initial approximation is critical for the speed of the algorithm. On the other hand, a seed with a large number of bits is costly to obtain. Consequently, finding a simple method of obtaining the seed value is of great practical interest in implementing the Newton-Raphson method. This speed-up method also applies in the Goldschmidt iteration for division and square root [4].

Direct lookup table implementations of such seeds are possible for small accuracy. For instance, the IBM 360/91 processor used generated 10-bit seed from a table (implemented as a ROM) addressed by 7 bits of the normalized divisor [2].

For larger accuracies, the bipartite method is more and more used [3]. For instance, in the AMD-K7 processor an optimized and simplified reciprocal and square root reciprocal estimate interpolation unit is implemented [8]. The reciprocal estimate provides at least 14.94 bits of accuracy while the square root reciprocal is accurate to at least 15.84 bits. This unit requires 69Kb of ROM and 3 cycles of latency.

1.2 Division Iteration

For the computation of the quotient $q = a/d$, one can use the following 2-step method:

1. Evaluate $t = 1/d$ using the Newton-Raphson iteration based on the function $f(x) = \frac{1}{x} - d$. The corresponding iteration is:

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - d}{-\frac{1}{x_i^2}} = x_i + x_i - dx_i^2 = x_i\left(2 - dx_i^2\right) \tag{3}$$

2. Compute the quotient using $q = t \times a$.

Each iteration requires two multiplications and one addition. The second step requires an additional full-length multiplication.

A table is often used to get the initial value x_0 based on a few most significant bits of d. As an example, on the Itanium processor, the `frcpa` instruction gives a seed for reciprocal $(1/d)$ with an accuracy of 8.886 bits (see [6] for details).

1.3 Square Root Iteration

Directly evaluating $\sqrt{c}$ using the Newton-Raphson algorithm with the function $f(x) = x^2 - c$ is not a good idea. The corresponding iteration is $x_{i+1} = \frac{1}{2}\left(x_i + \frac{c}{x_i}\right)$ which requires a division at each iteration. A better solution is based on the following 2-step method:

1. Newton-Raphson iteration based on the function $f(x) = \frac{1}{x^2} - c$ which has a zero equal to $\frac{1}{\sqrt{c}}$. The corresponding iteration is:

$$x_{i+1} = x_i - \frac{\frac{1}{x^2} - c}{-\frac{2}{x_i^3}} = x_i + \frac{x_i - cx_i^3}{2} = \frac{x_i}{2}\left(3 - cx_i^2\right) \tag{4}$$

2. Multiplication by c to get $\sqrt{c}$.

Each iteration requires three multiplications and one addition.

In this case also, a table is often used to get the initial value x_0 based on a few most significant bits of c. As an example, on the Itanium processor, the `frsqrta` instruction gives a seed for $1/\sqrt{c}$ with an accuracy of 8.831 bits (see [6] for details).

1.4 Minimax Polynomial Approximation

The initial approximations used in the following are based on the *minimax* polynomial approximation as a starting point. The degree-d minimax polynomial approximation to f on $[a, b]$ is the polynomial P^* that satisfies:

$$||f - P^*||_\infty = \min_{P \in \mathcal{P}_d} ||f - P||_\infty \tag{5}$$

where $\mathcal{P}_d$ is the set of polynomials with real coefficients and degree at most d and

$$||f - P||_\infty = \max_{a \le x \le b} |f(x) - P(x)|. \tag{6}$$

Minimax approximations can be computed using a well-known algorithm due to Remes [9] (available in the Maple numapprox package, for instance).

2 Reciprocal Seed

The degree-1 minimax polynomial of $f(x) = 1/x$ with $x \in [1, 2[$ (the range of the mantissa of a floating-point number) computed using Maple is

$$1.4571 - 0.5x \tag{7}$$

This polynomial provides an approximation with 4.5 bits of accuracy.

In this work, we use the minimax polynomial as a starting point because it is convenient and well implemented (e.g., Maple minimax function). But for degree-1 polynomial approximation, another method is possible to get the polynomial coefficients exactly, for details see [4, page 373]. Using this method one can get the polynomial $(3/4 + \sqrt{2}/2) - x/2$ which is theoretical polynomial that correspond to the minimax polynomial (7). Theoretically, this is the same polynomial, the slight difference in the constant coefficient is due to the rounding error during the Remez algorithm numerical execution. So, the two polynomials can be used as a starting point for our method since they provide similar accuracy and coefficients.

In the polynomial (7), the degree-1 coefficient is a power of 2. The constant coefficient is close to the value 1.5. So the following "close" polynomial can be used to approximate $1/x$:

$$p(x) = \frac{3}{2} - \frac{x}{2} \tag{8}$$

This modified polynomial p approximates $1/x$ on $[1, 2[$ with 3.5 bits of accuracy. We will use this very simple polynomial as a starting point to compute the seed value for the reciprocal function.

The evaluation cost of this polynomial is very small in practice. It is illustrated in Figure 1. Using 2's complement the value $-x/2$ is obtained by complementing all bits of x, shifted one position to the right, and adding a 1 in the LSB position. As $x \in [1, 2[$, the first bit of x is 1 (at position 0). If the binary representation of x is $(1.x_1x_2x_3 \ldots x_n)_2$, then the binary expansion of $-x/2$ is $(1.0\overline{x_1}\overline{x_2}\overline{x_3} \ldots \overline{x_n})_2$ plus 1 LSB. So the computation of $3/2 - x/2$ only costs one carry propagation corresponding to the additional 1 LSB. This kind of a very simple polynomial approximation was proposed in [5].

In order to improve the result of this approximation (p only provides 3.5 bits of accuracy), we add a *correcting term* $t(x)$ to the result of this modified polynomial. Our method is based on this very simple polynomial approximation (8) and a table to store the correcting terms. So the value of the seed is:

$$s(x) = p(x) + t(x) = \frac{3}{2} - \frac{x}{2} + t(x) \tag{9}$$

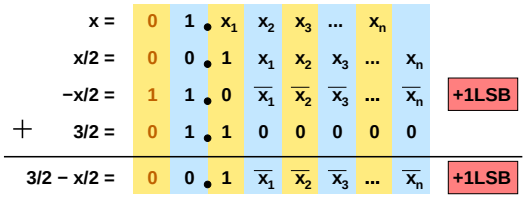

Figure 1: Efficient evaluation of the polynomial $p(x) = \frac{3}{2} - \frac{x}{2}$.

where the correcting term $t(x)$ is the difference between the actual function $1/x$ and the result of $p(x)$ rounded to the output width:

$$t(x) = \frac{1}{x} - \left(\frac{3}{2} - \frac{x}{2}\right) \tag{10}$$

The corresponding architecture is presented in Figure 3.

The table used to the correcting terms has n-bit addresses (the bits of the operand x) and w-bit words. The final result (seed) $s(x)$ has $n + g$ bits where g is the number of guard bits ($g > 1$ for the reciprocal function). The alignment of the n bits of p and the w bits of t is illustrated in Figure 2 using an example.

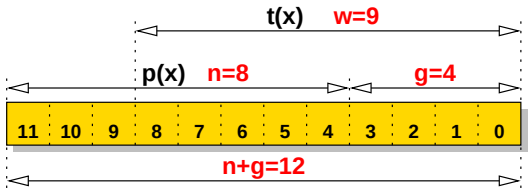

Figure 2: Alignment of p and t in the final result. For $n = 8$ and $g = 4$ the generator gives $w = 9$.

The main hardware cost of our solution is one $(2^n \times w)$-bit table and one $(n + g)$-bit addition with input carry. Optimizations at circuit level are presented in Section 4.

In addition to the VHDL description of the architecture, `seedgen` generates a plot of the approximation results. The corresponding plot is presented Figure 4 in the case $n = 3$ bits and $g = 1$ bit for the reciprocal function. The generated plot is based on a `gnuplot` script [1].

The accuracy (corresponding to the maximum error) provided by our architecture is $n + g + 1$ bits for the reciprocal. The choice of the correcting terms $t(x)$ can be done to ensure the seed value $s(x)$ is within a 1/2ulp distance to the theoretical value of $1/x$. The plot presented in Figure 5 illustrates the approximation error corresponding to the case $n = 3$ bits and $g = 1$ bit. This plot is also generated using `seedgen`.

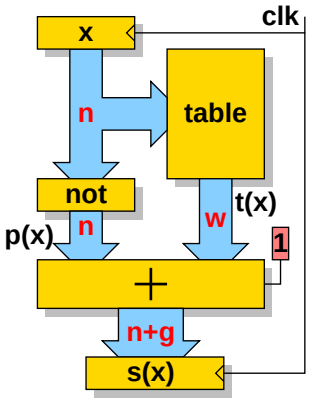

Figure 3: Architecture of the seed generator.

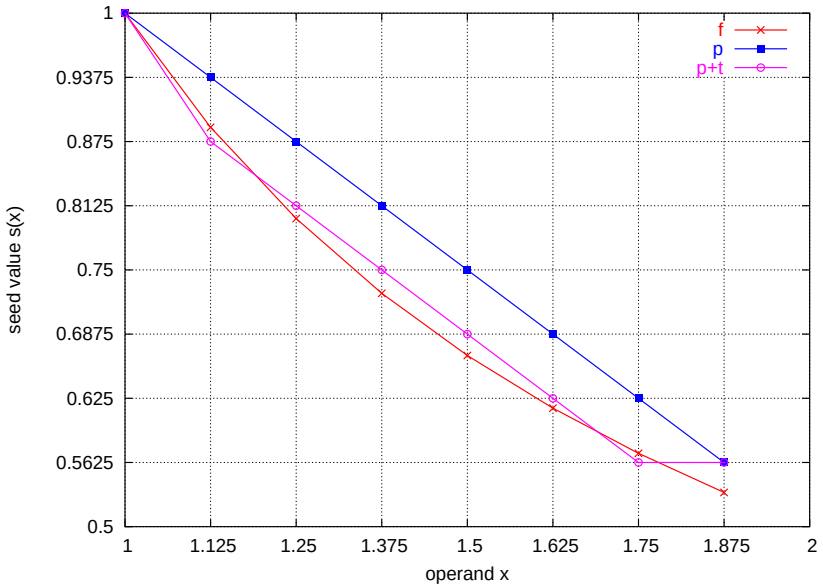

Figure 4: Approximation for $f = 1/x$, $n = 3$ and $g = 1$.

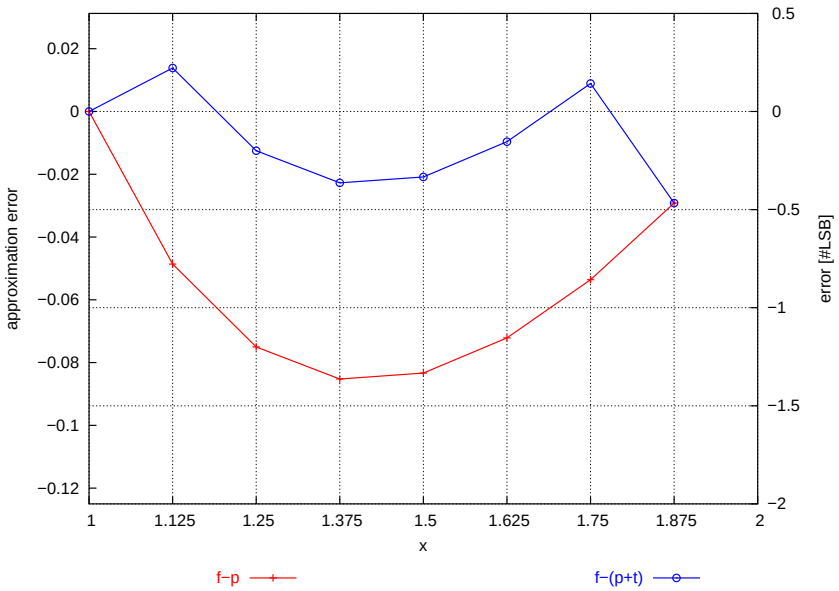

Figure 5: Approximation error for $f = 1/x$, $n = 3$ and $g = 1$.

The polynomial p overestimates the actual value of f (see Figure 4 for an illustration). Then the sign of the difference between p and f is always greater than or equal to zero. At the implementation level, the correction can be implemented using two solutions:

- store positive offsets in the table t and perform a subtraction $p - t$;
- store negative offsets in the table t and perform an addition $p + t$.

The best solution depends on the basic cells available at low level. Both solutions are possible using `seedgen` (see 4.3.1).

Figures 3 and 2 show that the alignment of p and t operands allow to improve the adder architecture used to perform $p + t$. Indeed, in some cases one can use incrementer cells instead of adder cells for the $w - g$ LSBs. In practice, this optimization is provided at the synthesis level by most of standard tools.

The detailed implementation results presented in Appendices A include a measure of this accuracy after the generation of the architecture.

3 Square Root Reciprocal Seed

A similar solution is proposed for the square root reciprocal. The degree-1 minimax polynomial of $f(x) = 1/\sqrt{x}$ with $x \in [1, 2[$, and 5.7 bits of accuracy is

$$1.2739 - 0.292x \tag{11}$$

This polynomial is close to:

$$p(x) = \frac{5}{4} - \frac{x}{4} \tag{12}$$

which gives an approximation to $1/\sqrt{x}$ on $[1, 2[$ with 4 bits of accuracy. The evaluation cost of this polynomial is also very small in practice as illustrated on Figure 6.

Using 2's complement the value $-x/4$ is obtained by complementing all bits of x shifted 2 bits to the right and add 1 LSB. As $x \in [1, 2[$, the binary expansion of $-x/4$ is $(1.10\overline{x_1 x_2 x_3} \ldots \overline{x_n})_2$ plus 1 LSB. The computation of $5/4 - x/4$ only costs one carry propagation corresponding the additional 1 LSB. So the same kind of architecture presented in Figure 3 can be used in case of the square root reciprocal.

x =	0	1	•	x_1	x_2	x_3	...	x_n			
x/4 =	0	0	•	0	1	x_1	x_2	x_3	...	x_n	
−x/4 =	1	1	•	1	0	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_3}$	...	$\overline{x_n}$	+1LSB
+ 5/4 =	0	1	•	0	1	0	0	0	0	0	
5/4 − x/4 =	0	0	•	1	1	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_3}$	...	$\overline{x_n}$	+1LSB

Figure 6: Efficient evaluation of the polynomial $p(x) = \frac{5}{4} - \frac{x}{4}$.

As the x is shifted two positions to the right the number of guard bits is $g \geq 2$ in the square root reciprocal case. Now the generator has to compute the correct constant bits of $-x/4$ and the correcting terms[1] $t(x)$ with

$$t(x) = \frac{1}{\sqrt{x}} - \left(\frac{5}{4} - \frac{x}{4}\right) \tag{13}$$

The accuracy (corresponding to the maximum error) provided by our architecture is also $n + g + 1$ bits (1/2 LSB) for the square root reciprocal. Figures 7 and 8 present respectively the approximation results plot and the error plot in the case $n = 3$ bits and $g = 1$ bit for the square root reciprocal function.

[1] The correcting terms t here are specific to the square root reciprocal functions.

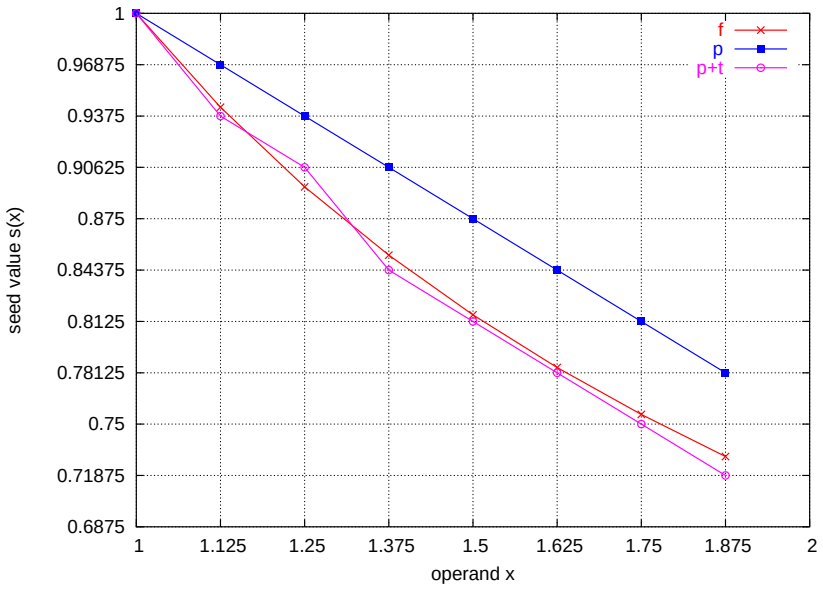

Figure 7: Approximation for $f = 1/\sqrt{x}$, $n = 3$ and $g = 2$.

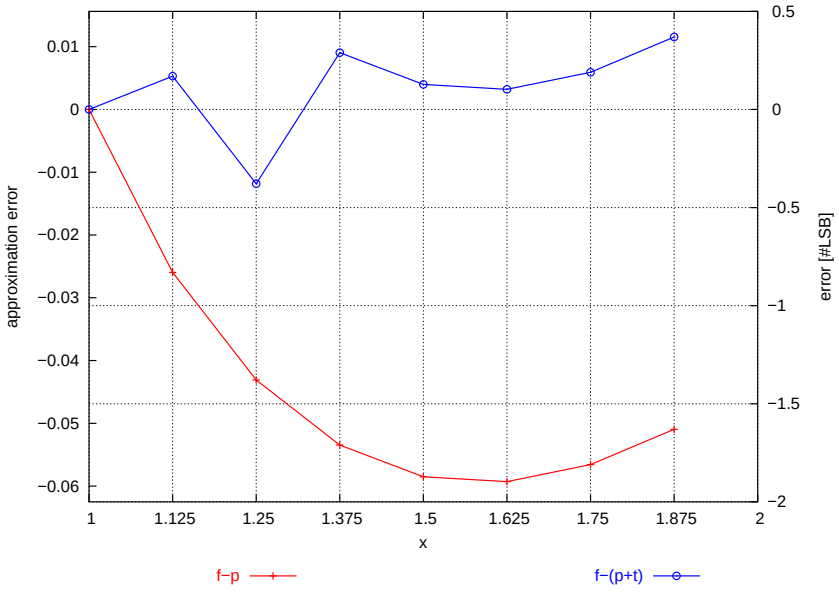

Figure 8: Approximation error for $f = 1/\sqrt{x}$, $n = 3$ and $g = 2$.

4 VHDL Generator

We have developed a program that generates optimized and synthesizable VHDL descriptions of seed architectures using our method. This program, called `seedgen`, is a C program in text mode and distributed under the GPL license. It is available on the Web [10].

4.1 Parameters Specification

All the `seedgen` parameters are given on the command line. Table 1 reports the main parameters.

parameter	meaning	value	
		possible	default
`-f`	approximated function (f)	`r` for $\frac{1}{x}$ `s` for $\frac{1}{\sqrt{x}}$	–
`-n`	argument size (n)	$2 \leq n \leq 16$	–
`-g`	number of guard bits (g)	$1 \leq g \leq 4$ for $\frac{1}{x}$ $2 \leq g \leq 4$ for $\frac{1}{\sqrt{x}}$	$g = 1$ for $\frac{1}{x}$ $g = 2$ for $\frac{1}{\sqrt{x}}$
`-o`	optimization	`n` for negative offset (see 4.3.1)	no optimization
`-r`	register	`o` output only `i` input only `b` both input and output	no register

Table 1: Main parameters supported by `seedgen`.

Below are examples of `seedgen` usage. The first one is for the reciprocal with 3-bit argument. The second one is for the square root reciprocal with 8-bit argument, registers at the input and the output of the operator and an optimization based on negative offsets in the table (see 4.3.1):

```
seedgen -f r -n 3

seedgen -f s -n 8 -r b -o n
```

4.2 Generated VHDL Code

The generated VHDL is restricted to synthesizable common declarations based on the IEEE 1164 library. An example of generated result is presented on Figure 9 and corresponds to the case of the reciprocal seed with the argument x on $n = 3$ bits ($g = 1$ (default value), no register and no optimization).

```
-- Generated by seedgen (version 0.1)
-- Thu Aug 18 13:40:36 2005
-- function: 1/x
-- n=3, g=1

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity seed is
  port(
       x : in std_logic_vector(2 downto 0);
       r : out std_logic_vector(4 downto 0)
      );
end seed;

architecture seedarch of seed is
  signal nx : std_logic_vector(2 downto 0);
  signal tbl : std_logic_vector(0 downto 0);
begin
  with x select
    tbl <= "0" when "000",   -- 0
           "1" when "001",   -- 1
           "1" when "010",   -- 1
           "1" when "011",   -- 1
           "1" when "100",   -- 1
           "1" when "101",   -- 1
           "1" when "110",   -- 1
           "0" when others;   -- 0
  nx <= not x;
  r <= ("01"&nx) - ("0000"&tbl) + "00001";
end seedarch;
```

Figure 9: Example of generated VHDL file.

The generation time is very small in practice. As an example, the reciprocal with $n = 12$ bits and all optimizations requires 0.9 seconds of generation of standard laptop (1.1 GHz) including the accuracy checking.

4.3 Supported Optimizations

At low level, the `not` block in Figure 3 depends on the implementation target. For instance, on some FPGAs the generated adder has complemented inputs (in that case the `not` block is virtual). The constant bits of x and $p(x)$ are propagated to the adder input using the synthesis tool.

4.3.1 Negative Offsets

As the value of the polynomial $p(x)$ are always larger (or equal) than the value of the target function $f(x)$, all the offsets stored in the table are less than zero (or equal in some cases). So the correcting terms can be implemented using two solutions:

- with positive correcting terms t and using a subtraction $p - t$ to compute the seed value,
- or with negative correcting terms and using an addition $p + t$.

The first solution (the default one) leads to two addition/subtraction operators and a $(2^n \times w)$-bit table. The second one leads to only one addition with carry-in and a $(2^n \times (w + 1))$-bit table. So there is a tradeoff between the addition/subtraction operator and the table size. As it is difficult to predict the actual result depending on the implementation target and the synthesis and place and route tools, the impact of the two solutions has to be tested in practice.

4.3.2 Additional Registers

It is sometimes useful to insert pipeline stages in the operator to speedup the circuit. The additional registers can be used by the synthesis tool in that way. Registers (with the correct size) can be added at the input and/or the output of the combinational operator.

5 Implementation Results

5.1 Accuracy Results

Our solution provides seeds (for reciprocal and square root reciprocal) with an accuracy of at most 1/2 LSB with words on $n + g$ bits. This corresponds to a maximum error of at most $2^{-(n+g+1)}$. In order to verify this accuracy, we present in Table 2 the minimum accuracy (corresponding to the maximum error) and the average accuracy (corresponding to

$f\downarrow$	$g\downarrow$	$n\rightarrow$	4	5	6	7	8	9	10	11	12
$\frac{1}{x}$	1	min.	6.04	7.02	8.01	9.00	10.00	11.00	12.00	13.00	14.00
		avg.	6.90	7.91	9.10	10.07	11.02	11.99	12.97	14.00	14.99
	2	min.	7.06	8.07	9.01	10.01	11.00	12.00	13.00	14.00	15.00
		avg.	8.16	9.24	10.05	11.10	12.02	12.98	14.00	15.01	16.01
	3	min.	8.07	9.03	10.01	11.00	12.00	13.00	14.00	15.00	16.00
		avg.	9.16	10.11	11.14	12.04	12.93	13.99	15.02	15.99	17.01
$\frac{1}{\sqrt{x}}$	2	min.	7.11	8.03	9.05	10.03	11.00	12.00	13.00	14.00	15.00
		avg.	8.02	8.87	10.15	10.98	12.00	12.98	13.98	15.02	16.02
	3	min.	8.03	9.05	10.05	11.01	12.00	13.00	14.00	15.00	16.00
		avg.	8.76	10.25	11.05	11.97	12.97	13.97	15.01	16.02	17.01
	4	min.	9.05	10.08	11.03	12.00	13.00	14.00	15.00	16.00	17.00
		avg.	10.27	11.16	11.95	12.87	13.97	15.00	15.99	17.00	17.99

Table 2: Error measurements (minimal and average accuracy) in number of correct bits.

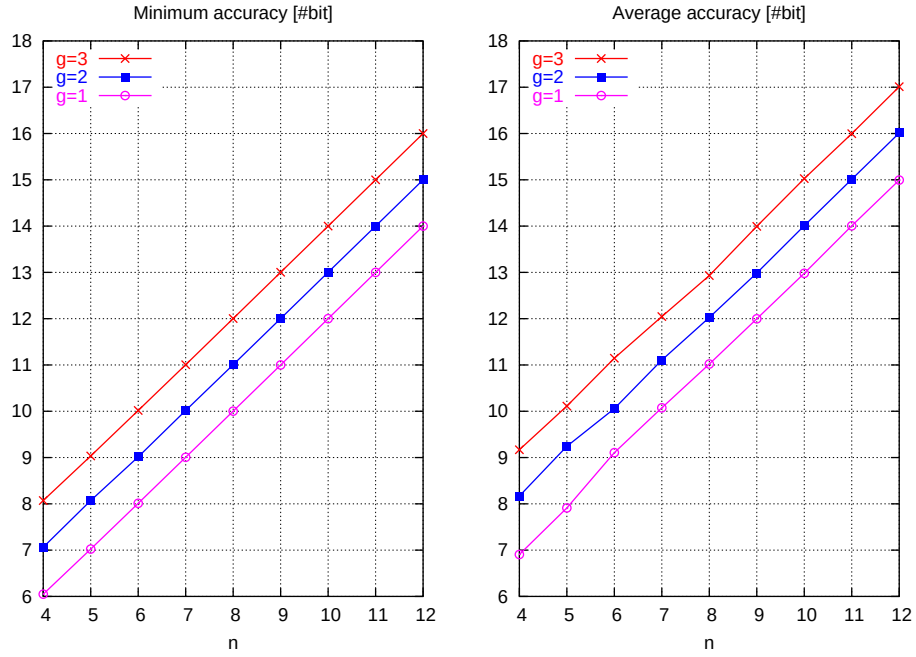

Figure 10: Maximum and average accuracy obtained for the reciprocal.

the average error) reported by the generator (using an exhaustive check). Figure 10 presents a plot of these results in case of the reciprocal.

Figure 11 presents the effective error for all the values of x in $[1, 2[$ and for several values of n. On this figure, three cases are presented: $n = 5$, $n = 6$ and $n = 7$ (all with $g = 1$). One can verify that the error is always less than 1/2LSB where the weight of the LSB is 2^{n+g}.

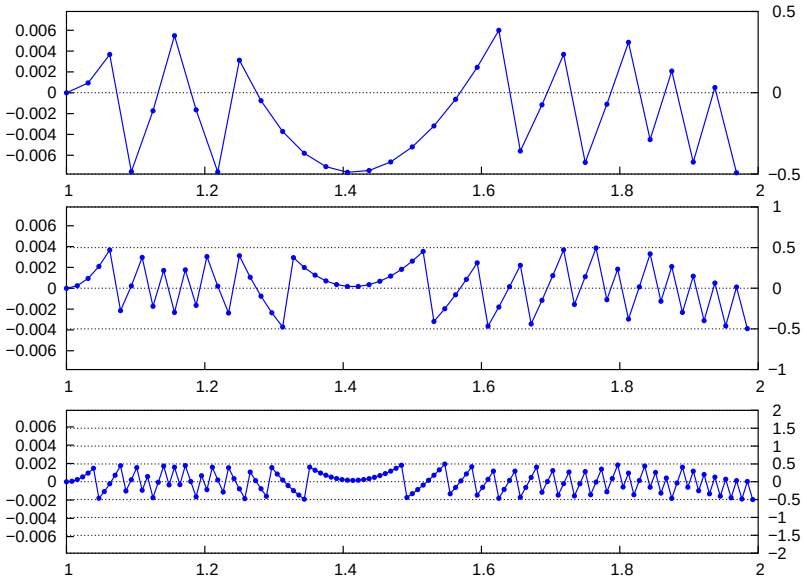

Figure 11: Evolution of the error for several values of n and the reciprocal ($g = 1$). Horizontal axis is x. Vertical axis is error (left scale) and error expressed in LSBs (right scale). Top curve is for $n = 5$ (32 points), middle curve for $n = 6$ (64 points) and bottom curve for $n = 7$ (128 points).

Figure 12 presents the effective error for all the values of x in $[1, 2[$ and for several values of g. On this figure, three cases corresponding to the reciprocal and $n = 6$ are presented: $g = 1$, $g = 2$ and $g = 3$ (all with $g = 1$). On this figure also, one can verify that the error is always less than 1/2LSB where there the weight of the LSB is 2^{n+g}.

5.2 Speed and Size Results

The complete FPGA implementation parameters and results are reported in Section A. Figure 13 presents the area and speed results on Spartan3 FPGAs (data extracted from Section A.1). In this figure, the horizontal axis represents the operand width n. The

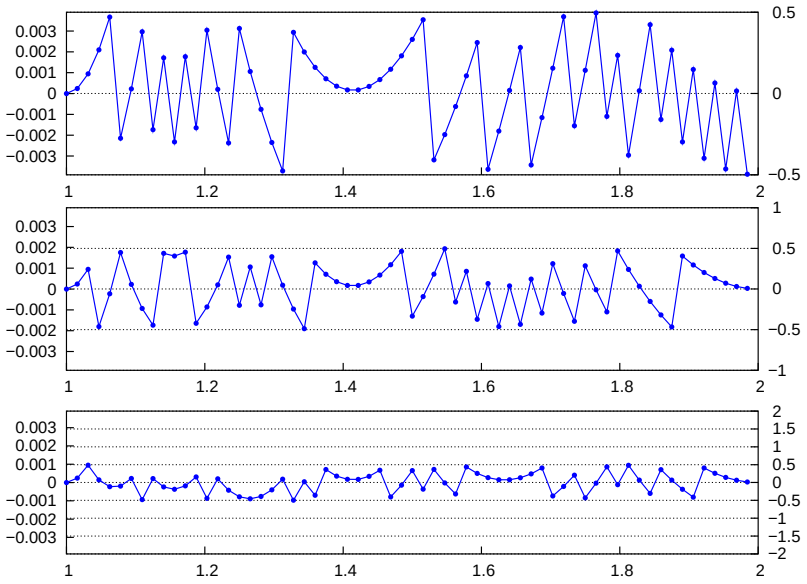

Figure 12: Evolution of the error for several values of g and the reciprocal ($n = 6$, i.e., 64 points). Horizontal axis is x. Vertical axis is error (left scale) and error expressed in LSBs (right scale). Top curve is for $g = 1$, middle curve for $g = 2$ and bottom curve for $g = 3$.

reported results here correspond to the optimization with negative offsets table as it usually gives the best results on FPGAs.

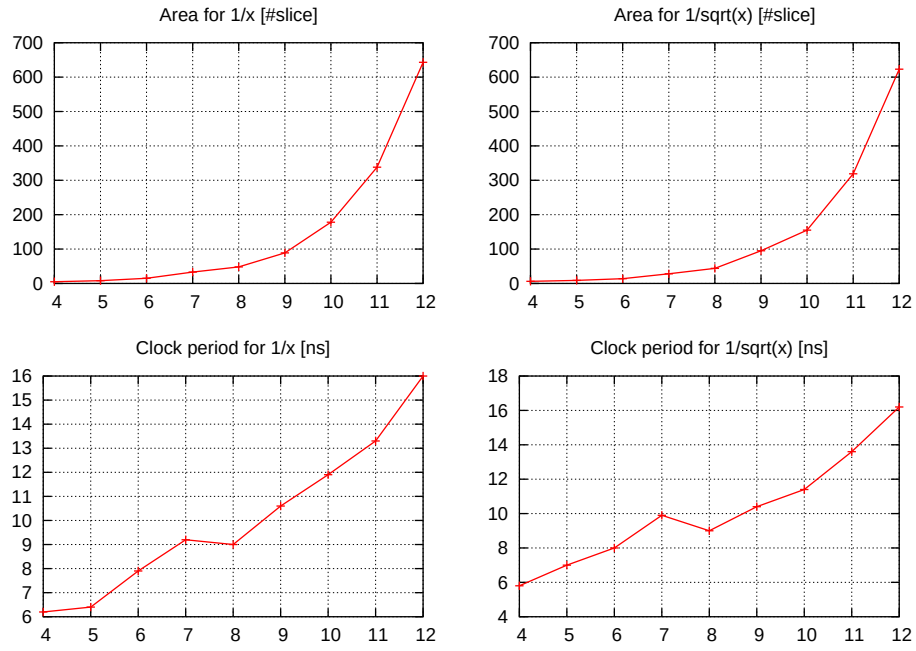

Figure 13: Implementation results on Spartan3 FPGA.

5.3 The Impact of Varying Parameters

As explained in Section 4.3.1, the correcting operation can be implemented using an addition or a subtraction depending on the offsets stored in the table. Figure 14 presents the area and speed of seeds for the reciprocal and several values of n. This figure shows that the negative offset optimization leads to faster circuits in FPGAs without area penalty.

The effort and the optimization target (area or speed) specified during the synthesis and the place and route processes may impact the performances of the circuit. Figure 15 presents the impact of the optimization target (area or speed) and the optimization effort (standard or high) on the implementation results. It shows that good results are obtained with area target and a standard effort. For an area target, a higher effort do not lead to better circuit. A speed effort (both with standard or high effort) may lead to faster circuits but at the cost of significantly larger areas.

For FPGA implementation, all the measurements shows that the best results are obtained for area optimization and the negative offset option (for reasonable values of n).

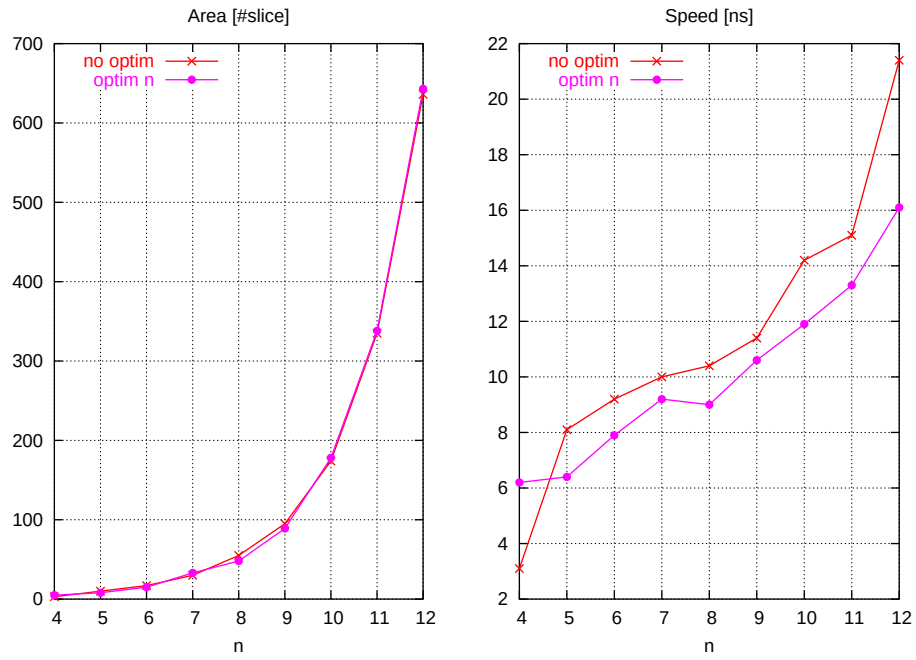

Figure 14: Impact of the negative offset optimization on the area and speed on FPGAs (for $f = 1/x$ and $g = 1$).

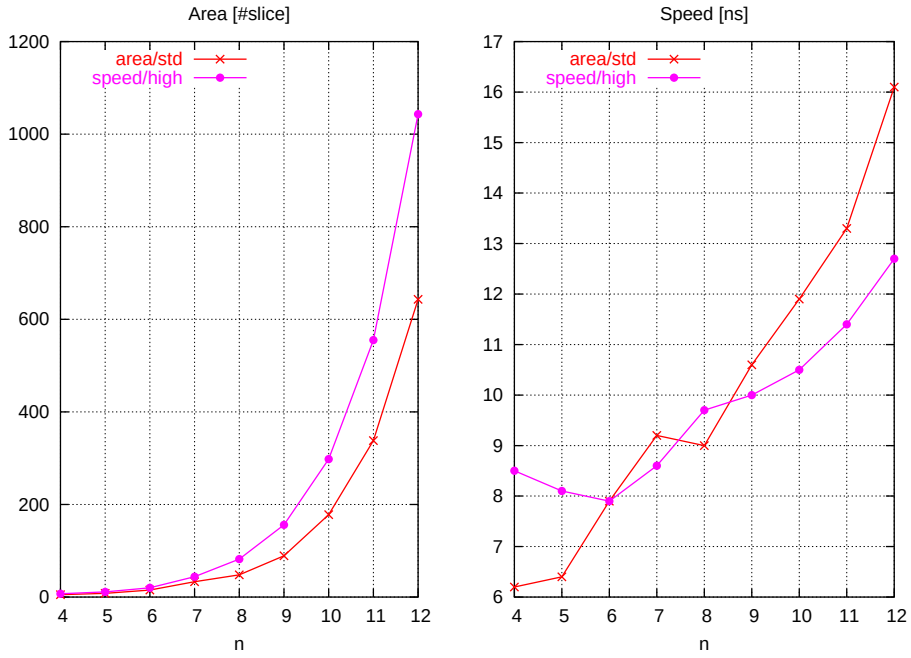

Figure 15: Impact of CAD tools options on the area and speed on FPGAs.

5.4 Comparison to Direct Lookup Table

In order to compare our method with standard solutions, we implemented seeds based on direct lookup table on FPGAs. The generated direct lookup table architecture is a ROM description with n-bit addresses and $n+g+1$-bit words. The stored values provide 1/2 LSB accuracy for all inputs as in operators generated by `seedgen`. When this ROM description is implemented on FPGAs, the synthesis tool performs a lot of logical optimization based on the content of the ROM. This leads to significantly smaller and faster architecture than using unoptimized table with $2^n \times (n+g+1)$ bits.

Table 3 presents the comparison results for the reciprocal on the Spartan3 FPGA and tools used in Appendix A. In this table, 3 solutions are compared. The first one (direct) is the ROM description without logical optimization (only area is reported). The second one (optimized) is the ROM description with the logical optimizations performed by the synthesis tool. A reduction factor less than 2 is obtained compared to the direct ROM description. The last solution is the results from `seedgen`. Our results show a reduction factor up to 2.5 and around 40% speed improvement compared to the optimized solution.

		direct		optimized		`seedgen`	
n	g	ROM size	area [#CLB]	area [#CLB]	period [ns]	area [#CLB]	period [ns]
7	2	1280	80	57	11.7	33	9.0
8	2	2816	176	109	12.7	48	9.2
9	2	6144	384	227	14.7	89	10.6
10	2	13312	832	448	16.4	178	11.9

Table 3: Comparison results between direct lookup table implementation (without and without logical optimization) and our method on Spartan3 FPGA.

Conclusion

A method for the initial approximation to reciprocal and square root reciprocal functions in hardware was proposed. These initial approximations or seeds are used in the initialization of floating-point division and square root software iterations in order to speedup the computation.

The proposed solution is based on polynomial approximation with specific coefficients and a table lookup. The obtained architectures lead to small and fast circuits. The method has been implemented in C program distributed under GPL license. This program, called `seedgen`, automatically generates optimized and synthesizable VHDL operators for various parameters and hardware constraints.

Compared to direct lookup table implementation on FPGAs (with content optimization using the synthesis tools), the proposed method leads to 2.5 reduction factor in size and 40% speed improvement.

A FPGA Implementation Results

Several operators generated by `seedgen` have been implemented on the Xilinx Spartan3 and VirtexII FPGAs. All the implementations have been performed using ISE 7.1i from Xilinx for synthesis as well as for the place and route steps. The results reported below have been obtained with an area optimization constraint and a standard effort for both synthesis and place and route.

In order to avoid the routing of long path in the input/output blocs of the FPGA, registers at the input and the output of the operator have been added.

A.1 Implementation Results on Spartan3 FPGA

The implementation results for the Spartan3 FPGA (xc3s400-ft256-4 with 3584 slices in this device) are reported in Tables 4 and 5 for the reciprocal and square root reciprocal respectively.

	accuracy [#bit]		table	optimization: none		optimization: `n`	
				area	period	area	period
n	minimal	average	size	[#slice]	[ns]	[#slice]	[ns]
4	6.0	6.9	$2^4 \times 2$	3	3.1	5	6.2
5	7.0	7.9	$2^5 \times 3$	10	8.1	8	6.4
6	8.0	9.1	$2^6 \times 4$	17	9.2	15	7.9
7	9.0	10.0	$2^7 \times 5$	30	10.0	33	9.2
8	10.0	11.0	$2^8 \times 6$	55	10.4	48	9.0
9	11.0	11.9	$2^9 \times 7$	95	11.4	89	10.6
10	12.0	12.9	$2^{10} \times 8$	174	14.2	178	11.9
11	13.0	14.0	$2^{11} \times 9$	335	15.1	338	13.3
12	14.0	14.9	$2^{12} \times 10$	636	21.4	643	16.1

Table 4: Implementation results for $\frac{1}{x}$ on Spartan3 FPGA.

A.2 Implementation Results on VirtexII FPGA

The implementation results for the VirtexII FPGA (xc2v500-fg256-5 with 3072 slices in this device) are illustrated in Figure 16 ($g = 1$ for $1/x$ and $g = 2$ for $1/\sqrt{x}$). In this figure, the

n	accuracy [#bit] minimal	accuracy [#bit] average	table size	optimization: none area [#slice]	optimization: none period [ns]	optimization: `n` area [#slice]	optimization: `n` period [ns]
4	7.1	8.0	$2^4 \times 3$	8	7.9	6	5.8
5	8.0	8.8	$2^5 \times 4$	13	8.5	9	7.0
6	9.0	10.1	$2^6 \times 4$	16	9.0	14	8.0
7	10.0	10.9	$2^7 \times 5$	28	9.9	28	9.9
8	11.0	12.0	$2^8 \times 6$	48	10.2	44	9.0
9	12.0	12.9	$2^9 \times 7$	93	11.2	95	10.4
10	13.0	13.9	$2^{10} \times 8$	163	15.3	155	11.4
11	14.0	15.0	$2^{11} \times 9$	325	15.7	319	13.6
12	15.0	16.0	$2^{12} \times 10$	616	19.5	623	16.2

Table 5: Implementation results for $\frac{1}{\sqrt{x}}$ on Spartan3 FPGA.

X axis represents the operand width n. The corresponding complete results are reported in Tables 6 and 7 for the reciprocal and square root reciprocal respectively.

n	accuracy [#bit] minimal	accuracy [#bit] average	table size	optimization: none area [#slice]	optimization: none period [ns]	optimization: `n` area [#slice]	optimization: `n` period [ns]
4	6.0	6.9	$2^4 \times 2$	3	3.2	5	5.3
5	7.0	7.9	$2^5 \times 3$	10	7.5	8	6.0
6	8.0	9.1	$2^6 \times 4$	17	8.6	15	6.9
7	9.0	10.0	$2^7 \times 5$	30	9.4	33	7.8
8	10.0	11.0	$2^8 \times 6$	55	10.0	48	8.4
9	11.0	11.9	$2^9 \times 7$	95	10.0	89	10.0
10	12.0	12.9	$2^{10} \times 8$	174	13.1	178	10.0
11	13.0	14.0	$2^{11} \times 9$	335	13.1	338	11.3
12	14.0	14.9	$2^{12} \times 10$	636	14.2	643	12.5

Table 6: Implementation results for $\frac{1}{x}$ on VirtexII FPGA.

One can notice that the area results for Spartan3 and VirtexII are similar. This is due to the fact that similar configurable logic block (CLB) resources are used in two FPGA families. But the speed of the VirtexII is greater than that of the Spartan3. For instance, in the case of the reciprocal function, with $n = 10$ and $g = 1$, the clock period is 11.9 ns for the Spartan3 and 10.0 ns for the VirtexII.

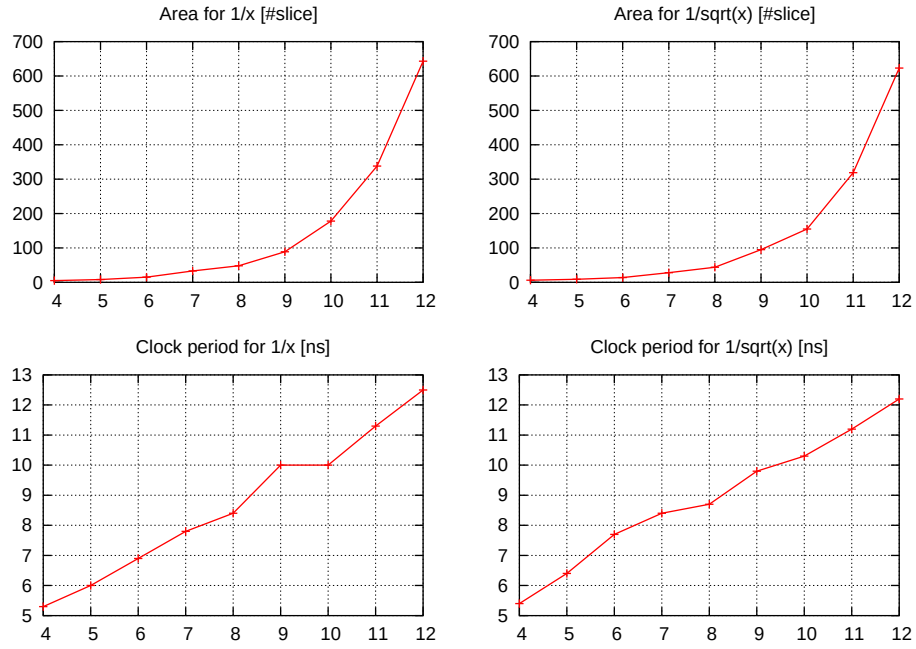

Figure 16: Implementation results on VirtexII FPGA.

n	accuracy [#bit] minimal	accuracy [#bit] average	table size	optimization: none area [#slice]	optimization: none period [ns]	optimization: `n` area [#slice]	optimization: `n` period [ns]
4	7.1	8.0	$2^4 \times 3$	8	7.2	6	5.4
5	8.0	8.8	$2^5 \times 4$	13	7.6	9	6.4
6	9.0	10.1	$2^6 \times 4$	16	8.4	14	7.7
7	10.0	10.9	$2^7 \times 5$	28	9.5	28	8.4
8	11.0	12.0	$2^8 \times 6$	48	9.7	44	8.7
9	12.0	12.9	$2^9 \times 7$	93	10.1	95	9.8
10	13.0	13.9	$2^{10} \times 8$	163	11.6	155	10.3
11	14.0	15.0	$2^{11} \times 9$	325	13.2	319	11.2
12	15.0	16.0	$2^{12} \times 10$	616	14.3	623	12.2

Table 7: Implementation results for $\frac{1}{\sqrt{x}}$ on VirtexII FPGA.

References

[1] Gnuplot. `http://www.gnuplot.info/`.

[2] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The IBM system/360 model 91: Floating-point execution unit. *IBM Journal*, 11(1):34–53, January 1967.

[3] D. Das Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In *12th IEEE Symposium on Computer Arithmetic*, pages 17–28. IEEE Computer Society, July 1995.

[4] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[5] M. Ito, N. Takagi, and S. Yajima. Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification. *IEEE Transactions on Computers*, 46(4):495–498, April 1997.

[6] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[7] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[8] S. F. Oberman. Floating point division and square root algorithms and implementation in the AMD-k7 microprocessor. In *14th IEEE Symposium on Computer Arithmetic*, pages 106–115. IEEE Computer Society, April 1999.

[9] E. Remes. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198:2063–2065, 1934.

[10] A. Tisserand. seedgen: A circuit generator for reciprocal and inverse square root seeds. `http://www.lirmm.fr/~tisseran/devel/seedgen`, GPL license, 2005.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399